Multi-Methods in Racket

António Menezes Leitão

April, 18, 2013

1 Introduction

Multi-methods are an advanced concept that extends the single dispatch approach that is used in the majority of object-oriented languages. Multi-methods were initially proposed for the CommonLoops language and were standardized in the Common Lisp Object System (CLOS). Many other languages have implemented multi-methods, either natively (e.g., Dylan, Cecil, Fortress, Clojure, etc), or with library extensions (e.g., C++, Python, Ruby, Java, etc).

Unfortunately, it is usually difficult to implement multi-methods in languages that were not designed to be extended, either because they have rigid programming paradigms or because they have a complex syntax, and this has been one of the biggest obstacles for the wider acceptance of multi-methods.

There are programming languages, however, that promote language extensions. The Racket programming language is one good example. Racket is a dialect of the Lisp family of languages that has many things in common with Scheme.

Racket was designed to support many different programming paradigms, including, among others, imperative programming, functional programming, and object-oriented programming. The language also supports both dynamically typed programs and statically typed programs, and it is possible to mix them. Finally, the language was designed to allow the implementation of additional programming paradigms, and to simplify its use via syntax extensions.

2 Goals

Implement, in Racket, the Racket Object System (ROS). ROS provides simplified versions of the generic functions and multi-methods available in CLOS. You must implement, at the very least, the features described in the following sections:

2.1 Generic Functions

You must implement syntax for the definition of a generic function, that only includes its name and the list of mandatory parameters. As an example, consider the following syntax for the definition of a generic function that "adds" things:

```
(defgeneric add (x y))
```

Suggestion: in order to implement generic functions, you will have to reify them, possibily, using Racket's `structs`. To extend Racket syntax to support generic function definitions, you will have to explore Racket's meta-programming capabilities, for example, `define-syntax-rule`.

2.2 Multi-Methods

You must implement syntax for the definition of a multi-method, including the name of the generic function to which it belongs, the list of parameters, and the body of the method. The list of parameters must be a list of lists, each sub-list containing two elements: the name of the parameter and a predicate (a function) that determines the applicability of the method in what regards that parameter.

As an example, consider the following syntax for the definition of two methods of the generic function `add`:

```
(defmethod add ((x number?) (y number?))
  (+ x y))

(defmethod add ((x string?) (y string?))
  (string-append x y))
```

Note, in the previous example, that `number?`, `string?`, `+`, and `string-append` are pre-defined functions in Racket.

Suggestion: use meta-programming to translate a method definition into a more complex expression that injects a representation of a method into the representation of a generic function.

2.3 Generic Function Call Protocol

You must implement a protocol for the application of a generic function to concrete arguments.

Given a generic function *gf* and a list of arguments *args*, this protocol should select the set of applicable methods of *gf*, sort them according to their specificity, and, finally, apply the most specific method to *args*. Note that a method is applicable to a set of arguments if each argument satisfies the predicate of the corresponding parameter. If there is no applicable method, an error must be raised with the expression `(error "Method missing for arguments" args)`.

As an example, considering the previous method definitions for the generic function `add`, the following interaction should be expected:

```
> (add 1 2)
3
> (add "Foo" "Bar")
"FooBar"
```

Suggestion: in order for the representation of a generic function to be treated as a function, you might want to represent it using a `struct` and also use the `prop:procedure` property of structures.

2.4 Method Ordering

You must implement a mechanism for sorting applicable methods. Precedence among applicable methods is determined by left-to-right consideration of parameter predicates. Method m_0 has higher precendence that method m_1 if the predicate of the first parameter of m_0 is more specific than the predicate of the first parameter of m_1. If they are **identical** predicates, then precedence is determined by the next parameter and so on.

Predicate specificity is not a trivial concept. To say that a predicate is more specific than other is equivalent to say that the set of values recognized by the first predicate is a subset of the values recognized by the second predicate. As an example, we can say that `integer?` is more specific than `number?` because all values that satisfy `integer?` also satisfy `number?` but not all values that satisfy `number?` also satisfy `integer?`.

Racket has many pre-defined predicates and the programmer can easily define additional ones. For this reason, the Racket language does not provide a direct way of testing the specificity of predicates, so you need to implement one.

As a concrete example, consider:

```
(defgeneric fact (n))

(defmethod fact ((n zero?))
  1)

(defmethod fact ((n integer?))
  (* n (fact (- n 1))))
```

In this example, the call `(fact 1)` has just one applicable method, but the recursive call `(fact 0)` has two applicable methods, because the argument `0` satisfies both predicates `integer?` and `zero?`. In order to sort the corresponding methods, it is necessary to assert that `zero?` is more specific than `integer?`. This is done using the function `defsubtype`, that accepts as arguments two functions and asserts the subtyping relation. The hierarchy of number types that is predefined in Racket, suggests, at the very least, the following relations:

```
(defsubtype complex? number?)
(defsubtype real? complex?)
(defsubtype rational? real?)
(defsubtype integer? rational?)
(defsubtype zero? integer?)
```

Note that, by transitivity, it should be possible to derive, e.g., that `zero?` is a subtype of `number?`.

Obviously, more fine-grained relationships can be defined. For example, instead of having

```
(defsubtype zero? integer?)
```

we can have instead

```
(defsubtype odd? integer?)
(defsubtype even? integer?)
(defsubtype zero? even?)
```

Using the `defsubtype` relation, it becomes possible to define the specificity of predicates that is used to order applicable methods.

However, it might not always be possible to establish the specificity among two predicates. As an example, consider the predicates `integer?` and `positive?` and the following generic function:

```
(defgeneric what-are-you? (x))

(defmethod what-are-you? ((x integer?))
  "an integer")

(defmethod what-are-you? ((x positive?))
  "a positive number")
```

The previous example is ambiguous because some, but not all, integers are positive and some, but not all, positive numbers are integers. In order to be deterministic, we will assume that when it is not possible to determine the specificity of predicates, method ordering relies instead on the definition order: if method m_0 is defined **before** method m_1, then m_0 is **more specific** than m_1. This reflects the typical pattern where we take care of the most specific cases first. In the previous example, this means that the query `(what-are-you? 1)` should return `"an integer"`.

As another example, we will (incorrectly) define the generic function `length` but assuming that there is no predefined subtyping relation between the predicates `list?` and `null?`:

```
(defgeneric length (l))

(defmethod length ((l list?))
  (+ 1 (length (rest l))))

(defmethod length ((l null?))
  0)
```

Using the above definition order, the call `(length '(1 2 3 4))` triggers an error because the second method, that stops the recursion, is never called. In order to solve the problem, either the definition order is reversed, or we include the relation `(defsubtype null? list?)`.

Suggestion: keep the methods of a generic function sorted according to the definition order and then use the Racket function `sort` that implements *stable sorting* to sort between applicable methods using the specificity relation.

2.5 Dynamic Method Definition

Generic functions should be dynamic, in the sense that it should always be possible to define additional methods at run-time. As an example, consider the following function that computes the power function in a linear number of steps:

```
(defgeneric power (x n))

(defmethod power ((x number?) (n integer?))
  (* x (power x (- n 1))))

(defmethod power ((x number?) (n zero?))
  1)
```

After verifying that the function consumes way too much time, we might decide to speed it up by injecting an additional method that makes a significant fraction of the process to run in a logarithmic number of steps:

```
(defmethod power ((x number?) (n even?))
  (let ((x^n/2 (power x (/ n 2))))
    (* x^n/2 x^n/2)))
```

In order for this last method to have any effect, it is necessary to be able to dynamically update the generic function.

2.6 Dynamic Method Redefinition

It should be possible to redefine a method of a generic function. To this end, you should treat two methods with the exact same sequence of parameter predicates as the same method. This can then be used to replace an old definition with a new one.

This means that the following example should correctly compute the factorial of an integer:

```
(defgeneric fact (n))

(defmethod fact ((n zero?))
  0)

(defmethod fact ((n integer?))
  (* n (fact (- n 1))))

(defmethod fact ((n zero?))
  1)
```

2.7 Reflective Operations

You need to implement reflective operations over generic functions and methods, including:

- `generic-function-parameters` that accepts a generic function and returns the list of parameters of the function.
- `generic-function-methods` that accepts a generic function and returns the list of methods of the function.
- `method-types` that accepts a method and returns the list of predicates used in the method parameter list.

As an example, consider:

```
> fact
#<procedure:generic-function>
> (generic-function-parameters fact)
'(n)
> (generic-function-methods fact)
'(#<method> #<method>)
> (map method-types (generic-function-methods fact))
'((#<procedure:zero?>) (#<procedure:integer?>))
```

Suggestion: Racket's `struct`s automatically define functions to obtain the values of the fields. A good choice of names for these fields can make this task extremely simple.

2.8 Extensions

You can extend your project to further increase your grade. Note that this increase will not exceed **two** points that will be added to the project grade for the implementation of what was required in the other sections of this specification.

Be careful when implementing extensions, so that extra functionality does not compromise the functionality asked in the previous sections.

Some of the potentially interesting extensions include:

- Allowing generic functions to accept an argument precedence order similar to the one used in CLOS
- Allowing methods to have roles (e.g., before, after, and around) and generic functions to provide the standard method combination available in CLOS
- Allowing the subtyping relation to create a graph of relations (and not only a tree)
- Allowing generic functions to implement the simple method combination available in CLOS
- Implementation of a class system with inheritance that automatically establishes the subtyping relation.
- Providing a meta-object protocol that allows subclasses of generic functions and methods to have different behavior.

3 Code

Your implementation must work in Racket, version 5.3.1.

The written code should have the best possible style, should allow easy reading and should not require excessive comments. It is always preferable to have clearer code with few comments than obscure code with lots of comments.

The code should be modular, divided in functionalities with specific and reduced responsibilities. Each module should have a short comment describing its purpose.

You must `provide` all the names requested in this specification, namely:

- `defgeneric`
- `defmethod`
- `defsubtype`
- `generic-function-parameters`
- `generic-function-methods`
- `method-types`

4 Presentation

For this project, a full report is not required. Instead, a public presentation is required. This presentation should be prepared for a 15 minutes slot (approximately, 8 slides), should be centered in the architectural decisions taken and might include all the details that you consider relevant. You should be able to "sell" your solution to your colleagues and teachers.

5 Format

Each project must be handled by electronic means using the Fénix Portal. Each group must handle a single compressed file in ZIP format, named as `ros.zip`. This file must contain:

- the source code,
- the slides of the presentation,
- a `ros.rkt` file that, if needed, requires all the necessary files, and provides the names needed to use the system

The accepted format for the presentation slides is PDF. This file must be located at the root of the ZIP file and must have the name `p2.pdf`.

The file `ros.rkt` is a Racket file containing all the code necessary to implement ROS, the Racket Object System, as described previously. This means that starting from an operating system shell, the following interaction should be possible:

```
$ racket
Welcome to Racket v5.1.3.
> (require "ros.rkt")
> (defgeneric foo (x))
> (defmethod foo ((x integer?)) (+ x 1))
> (foo 1)
2
```

6 Evaluation

The evaluation criteria include:

- The quality of the developed solutions.
- The clarity of the developed programs.
- The quality of the public presentation.

In case of doubt, the teacher might request explanations about the inner working of the developed project, including demonstrations.

7 Plagiarism

It is considered plagiarism the use of any fragments of programs that were not provided by the teachers. It is not considered plagiarism the use of ideas given by colleagues as long as the proper attribution is provided.

This course has very strict rules regarding what is plagiarism. Any two projects where plagiarism is detected will receive a grade of zero.

These rules should not prevent the normal exchange of ideas between colleagues.

8 Final Notes

Don't forget Murphy's Law.

9 Deadlines

The code and the slides must be handled via Fénix, no later than 20:00 of **May, 10**.

The presentations will be done during the classes after the deadline. Only one element of the group will present the work and the presentation must not exceed 15 minutes. The element will be chosen by the teacher just before the presentation.